LL Grammars

Mattox Beckman

beckman@iit.edu

Illinois Institute of Technology

The topic for this lecture is a kind of grammar that works well with recursive-descent parsing.

- Know how to tell if a grammar is LL.
- Know what parsing technique will work with an LL grammar.
- Know how to detect and eliminate left recursion.
- Know how to detect and eliminate common prefixes.

Further reading: See Dragon Book §4.x

- An LL parse uses a **L**eft-to-right scan and produces a **L**eftmost derivation, using **n** tokens of lookahead.
- A.K.A. Top-Down Parsing

Example Grammar:

$S \rightarrow$ `+` $E\ E$
$E \rightarrow$ `int`
$E \rightarrow$ `*` $E\ E$

Example Input: `+ 2 * 3 4`

Syntax Tree:

S

Derivations:

S

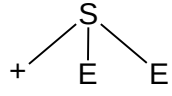
- An LL parse uses a **L**eft-to-right scan and produces a **L**eftmost derivation, using **n** tokens of lookahead.
- A.K.A. Top-Down Parsing

Example Grammar:

$S \rightarrow$ `+` $E\ E$
$E \rightarrow$ `int`
$E \rightarrow$ `*` $E\ E$

Example Input: `+ 2 * 3 4`

Syntax Tree:



Derivations:

S

+ E E

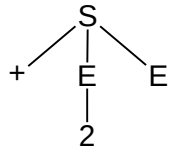
- An LL parse uses a **L**eft-to-right scan and produces a **L**eftmost derivation, using **n** tokens of lookahead.
- A.K.A. Top-Down Parsing

Example Grammar:

$S \rightarrow$ + E E
$E \rightarrow$ int
$E \rightarrow$ * E E

Example Input: + 2 * 3 4

Syntax Tree:



Derivations:

S
+ E E
+ 2 E

LL Grammars
What is LL(n) Parsing?
February 12, 2007
§0 **Objectives**

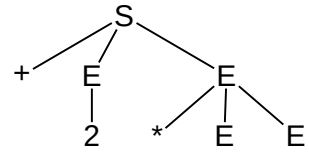
- An LL parse uses a **L**eft-to-right scan and produces a **L**eftmost derivation, using **n** tokens of lookahead.
- A.K.A. Top-Down Parsing

Example Grammar:

$S \rightarrow$ + E E
$E \rightarrow$ int
$E \rightarrow$ * E E

Example Input: + 2 * 3 4

Syntax Tree:



Derivations:

S
+ E E
+ 2 E
+ 2 * E E

Illinois Institute of Technology Mattox Beckman
LL Grammars – p. 3

LL Grammars
What is LL(n) Parsing?
February 12, 2007
§0 **Objectives**

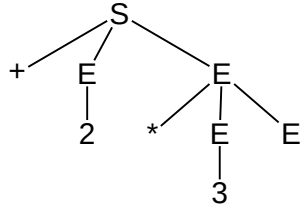
- An LL parse uses a **L**eft-to-right scan and produces a **L**eftmost derivation, using **n** tokens of lookahead.
- A.K.A. Top-Down Parsing

Example Grammar:

$S \rightarrow$ + E E
$E \rightarrow$ int
$E \rightarrow$ * E E

Example Input: + 2 * 3 4

Syntax Tree:



Derivations:

S
+ E E
+ 2 E
+ 2 * E E
+ 2 * 3 E

Illinois Institute of Technology Mattox Beckman
LL Grammars – p. 3

LL Grammars
What is LL(n) Parsing?
February 12, 2007
§0 **Objectives**

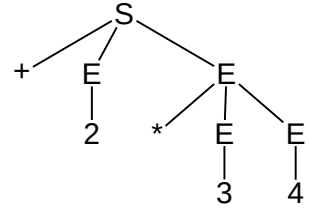
- An LL parse uses a **L**eft-to-right scan and produces a **L**eftmost derivation, using **n** tokens of lookahead.
- A.K.A. Top-Down Parsing

Example Grammar:

$S \rightarrow$ + E E
$E \rightarrow$ int
$E \rightarrow$ * E E

Example Input: + 2 * 3 4

Syntax Tree:



Derivations:

S
+ E E
+ 2 E
+ 2 * E E
+ 2 * 3 E
+ 2 * 3 4

Illinois Institute of Technology Mattox Beckman
LL Grammars – p. 3

How to Implement It

LL Parsers are very easy to implement.

- Consider each non-terminal rule to be a function.
- Each terminal consumes an input.
- Each function has type `string list -> tree * string list`
  - input is a list of tokens
  - output is a syntax tree and remaining tokens.
- You also need to create a type to represent your tree.

```
# getS ["+"; "2"; "*"; "3"; "4"];;
- : s * string list =
   S (Eint 2, Etimes (Eint 3, Eint 4)), []
```


Things to Notice

- Each function immediately checks the first token of the input string. (LL(0) parsers will consume the token immediately.)
- This token is used to decide what to do next.

E → int | * E E

```
let rec getE input =
  match input with
  | [] -> raise ParseError
  | "*"::xs -> let (e1,rest) = getE xs in
               let (e2,rest) = getE rest in
                 (Etimes (e1,e2), rest)
  |  i::xs -> (Eint (int_of_string i), xs)
```

- What kinds of things could go wrong?

Left Recursion

- A rule like $E \rightarrow E + E$ would cause an infinite loop.

E → E + E

```
let rec getE input =
  match input with
  | [] -> raise ParseError
  | _ -> let (e1,rest) = getE input in
         let ["+";rest] = rest in
         let (e2,rest) = getE rest in
             (Eplus (e1,e2), rest)
```


Rules with Common Prefixes

- A rule like $E \rightarrow - E \mid - E E$ would confuse the function. Which version of the rule should be used?

```
let rec getE input =
  match input with
  | [] -> raise ParseError
  | "-"::xs -> let (e1,rest) = getE xs in
                 (Enegative e1, rest)
  | "-"::xs -> let (e1,rest) = getE xs in
               let (e2,rest) = getE rest in
                 (Eminus (e1,e2), rest)
```

Consider deriving `i++++` from the following grammar:

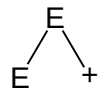
$E \rightarrow E +$
$E \rightarrow i$

Derivation:

E

Consider deriving `i++++` from the following grammar:

$E \rightarrow E +$
$E \rightarrow i$

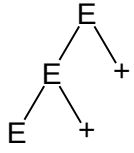


Derivation:

E
E+

Consider deriving `i++++` from the following grammar:

$E \rightarrow E +$
$E \rightarrow i$

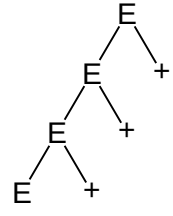


Derivation:

E
E+
E++

Consider deriving `i++++` from the following grammar:

$E \rightarrow E +$
$E \rightarrow i$

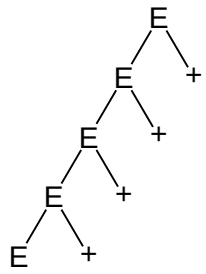


Derivation:

E
E+
E++
E+++

Consider deriving `i++++` from the following grammar:

$E \rightarrow E +$
$E \rightarrow i$



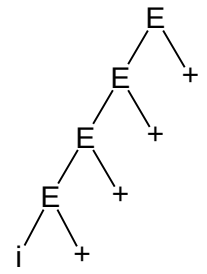
Derivation:

E
E+
E++
E+++
E++++

LL Grammars February 12, 2007
The Idea §2 **Eliminating Left Recursion**

Consider deriving `i++++` from the following grammar:

$E \rightarrow E +$
$E \rightarrow i$



Derivation:

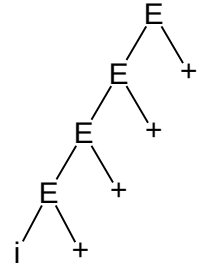
E
E+
E++
E+++
E++++
i++++

Illinois Institute of Technology Mattox Beckman LL Grammars – p. 8

LL Grammars February 12, 2007
The Idea §2 **Eliminating Left Recursion**

Consider deriving `i++++` from the following grammar:

$E \rightarrow E +$
$E \rightarrow i$



Derivation:

E
E+
E++
E+++
E++++
i++++

- The rule $E \rightarrow E+$ says that we can have as many +s as we want *at the end* of the sentence.
- The rule $E \rightarrow i$ says—in effect—that the first word can be a `i`.
- Question: isn't there another way to write this?

Illinois Institute of Technology Mattox Beckman LL Grammars – p. 8

LL Grammars February 12, 2007
More complicated example §2 **Eliminating Left Recursion**

Consider the following grammar. What does it mean?

$$B \rightarrow Bxy \mid Bz \mid q \mid r$$

- At the end can come any combination of x y or z
- At the beginning can come q or r

Eliminating the Left Recursion

We can rewrite these grammars $E \to E + \mid i$
$B \to Bxy \mid Bz \mid q \mid r$
using the following transformation:

- Productions of the form $S \to \beta$ become $S \to \beta S'$.
- Productions of the form $S \to S\alpha$ become $S' \to \alpha S'$.
- Add $S' \to \epsilon$.

Result:
$E \to iE'$
$E' \to +E' \mid \epsilon$
$B \to qB' \mid rB'$
$B' \to xyB' \mid zB' \mid \epsilon$

Mutual Recursions!

Things are slightly more complicated if we have mutual recursions.

$$A \to Aa \mid Bb \mid Cc \mid q$$
$$B \to Ax \mid By \mid Cz \mid rA$$
$$C \to Ai \mid Bj \mid Ck \mid sB$$

How to do it:

- Take the first symbol (A) and eliminate immediate left recursion.
- Take the second symbol (B), and substitute left recursions to A. Then eliminate immediate left recursion in B.
- Take the third symbol (C) and substitute left recursions to A and B. Then eliminate immediate left recursion in C.

Left Recursion Example

Here is a more complex left recursion.
$A \to Aa \mid Bb \mid Cc \mid q$
$B \to Ax \mid By \mid Cz \mid rA$
$C \to Ai \mid Bj \mid Ck \mid sB$
First we eliminate the left recursion from A.
$A \to Aa \mid Bb \mid Cc \mid q$
becomes
$A \to BbA' \mid CcA' \mid qA'$
$A' \to aA' \mid \epsilon$

Left Recursion Example, 2

We substituting in the new definition of A, and now we will work on the B productions.
$A \to BbA' \mid CcA' \mid qA'$
$A' \to aA' \mid \epsilon$
$B \to Ax \mid By \mid Cz \mid rA$
$C \to Ai \mid Bj \mid Ck \mid sB$
First, we eliminate the "backward" recursion from B to A.
$B \to Ax$ becomes
$B \to BbA'x \mid CcA'x \mid qA'x$

Left Recursion Example, 3

$A \to BbA' \mid CcA' \mid qA'$
$A' \to aA' \mid \epsilon$
$B \to BbA'x \mid CcA'x \mid qA'x \mid By \mid Cz \mid rA$
$C \to Ai \mid Bj \mid Ck \mid sB$

Now we can eliminate the simple left recursion in B, to get

$B \to CcA'xB' \mid qA'xB' \mid CzB' \mid rAB'$
$B' \to bA'xB' \mid yB' \mid \epsilon$

Left Recursion Example, 4

$A \to BbA' \mid CcA' \mid qA'$
$A' \to aA' \mid \epsilon$
$B \to CcA'xB' \mid qA'xB' \mid CzB' \mid rAB'$
$B' \to bA'xB' \mid yB' \mid \epsilon$
$C \to Ai \mid Bj \mid Ck \mid sB$

Now production C: first, replace left recursive calls to A...

$$C \to BbA'i \mid CcA'i \mid qA'i \mid Bj \mid Ck \mid sB$$

Next, replace left recursive calls to B (this gets messy)...

$$\begin{aligned} C \to\ & CcA'xB'bA'i \mid qA'xB'bA'i \mid CzB'bA'i \mid rAB'bA'i \\ & CcA'xB'j \mid qA'xB'j \mid CzB'j \mid rAB'j \\ & CcA'i \mid qA'i \mid Ck \mid sB \end{aligned}$$

Left Recursion Example, 5

Reorganizing C, we have

$$\begin{aligned} C \to\ & qA'xB'bA'i \mid rAB'bA'i \mid qA'xB'j \mid rAB'j \mid qA'i \mid sB \\ & CcA'xB'bA'i \mid CzB'bA'i \mid CcA'xB'j \mid CzB'j \mid CcA'i \mid Ck \end{aligned}$$

Eliminating left recursion gives us

$$\begin{aligned} C \to\ & qA'xB'bA'iC' \mid rAB'bA'iC' \mid qA'xB'jC' \mid rAB'jC' \mid qA'iC' \mid sBC' \\ C' \to\ & cA'xB'bA'iC' \mid zB'bA'iC' \mid cA'xB'jC' \mid zB'jC' \mid cA'iC' \mid kC' \mid \epsilon \end{aligned}$$

The result...

Our final grammar is now

$A \to BbA' \mid CcA' \mid qA'$
$A' \to aA' \mid \epsilon$
$B \to CcA'xB' \mid qA'xB' \mid CzB' \mid rAB'$
$B' \to bA'xB' \mid yB' \mid \epsilon$
$C \to qA'xB'bA'iC' \mid rAB'bA'iC' \mid qA'xB'jC' \mid rAB'jC' \mid qA'iC' \mid sBC'$
$C' \to cA'xB'bA'iC' \mid zB'bA'iC' \mid cA'xB'jC' \mid zB'jC' \mid cA'iC' \mid kC' \mid \epsilon$

Beautiful, isn't it? I wonder why we don't do this more often?

- Disclaimer: if there is a cycle ($A \to^+ A$) or an epsilon production ($A \to \epsilon$) then this technique is not guaranteed to work.

This grammar has common prefixes.

$$A \to xyB \mid CyC \mid q$$
$$B \to zC \mid zx \mid w$$
$$C \to y \mid x$$

To check for common prefixes, take a non-terminal and compare the First sets of each production.

Production	FirstSet
$A \to xyB$	$\{x\}$
$A \to CyC$	$\{x, y\}$
$A \to q$	$\{q\}$

If we are viewing an A, we will want to look at the next token to see which A production to use. If that token is x, then which production do we use?

If $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \gamma$ we can rewrite it as $A \to \alpha A' \mid \gamma$, $A' \to \beta_1 \mid \beta_2$

So, in our example:

$A \to xyB \mid CyC \mid q$
$B \to zC \mid zx \mid w$
$C \to y \mid x$

becomes

$A \to xA' \mid q \mid yyC$
$A' \to yB \mid yC$
$B \to zB' \mid w$
$B' \to C \mid x$
$C \to y \mid x$

Sometimes you'll need to do this more than once. Note that this process can destroy the meaning of the nonterminals.

One of these is LL, the other two are not. Fix the ones that are not.

Grammar 1

$E \to E\ x\ y$
$E \to E\ x\ B$
$E \to q$
$B \to E\ z$

Grammar 2

$S \to A\ x$
$S \to B\ y$
$A \to z\ B$
$B \to w\ A$

Grammar 3

$S \to A\ x$
$S \to B\ y$
$A \to z\ B$
$B \to z$

Grammar 1 starts as:

$E \to E\ x\ y$
$E \to E\ x\ B$
$E \to q$
$B \to E\ z$

Eliminate the left-recursion to get:

$E \to qE'$
$E' \to x\ y\ E'$
$E' \to x\ B\ E'$
$E' \to \epsilon$
$B \to E\ z$

Fix the common prefixes to get....

$E \to qE'$
$E' \to x\ E''$
$E' \to \epsilon$
$E'' \to y\ E' \mid B\ E'$
$B \to E\ z$

Grammar 2

$$S \rightarrow A\ x$$
$$S \rightarrow B\ y$$
$$A \rightarrow z\ B$$
$$B \rightarrow w\ A$$

It doesn't terminate. But it's not left recursive, and it has no productions with common prefixes, so it's still LL.

Grammar 3

$$S \rightarrow A\ x$$
$$S \rightarrow B\ y$$
$$A \rightarrow z\ B$$
$$B \rightarrow z$$

Production S has common prefixes. One way to fix this grammar is to eliminate the distinctions between A and B — this may not be what you want, though.

$$S \rightarrow z\ S'$$
$$S' \rightarrow z\ x$$
$$S' \rightarrow y$$